Týr: Blob Storage Meets Built-In Transactions

Pierre Matri*, Alexandru Costan†, Gabriel Antoniu‡, Jesús Montes*, María S. Pérez*

*Ontology Engineering Group, Universidad Politécnica de Madrid, Madrid, Spain, {pmatri, jmontes, mperez}@fi.upm.es
†IRISA / INSA Rennes, Rennes, France, alexandru.costan@irisa.fr
‡Inria Rennes Bretagne-Atlantique, Rennes, France, gabriel.antoniu@inria.fr

***Abstract*—Concurrent Big Data applications often require high-performance storage, as well as ACID *(Atomicity, Consistency, Isolation, Durability)* transaction support. Although blobs (binary large objects) are an increasingly popular storage model for such applications, state-of-the-art blob storage systems offer no transaction semantics. This demands users to coordinate data access carefully in order to avoid race conditions, inconsistent writes, overwrites and other problems that cause erratic behavior. We argue there is a gap between existing storage solutions and application requirements, which limits the design of transaction-oriented applications. We introduce Týr, the first blob storage system to provide built-in, multiblob transactions, while retaining sequential consistency and high throughput under heavy access concurrency. Týr offers fine-grained random write access to data and in-place atomic operations. Large-scale experiments with a production application from CERN LHC show Týr throughput outperforming state-of-the-art solutions by more than 75%.**

I. Introduction

Binary Large Objects (or *Blobs*) are an increasingly popular storage model for data-intensive applications, in which the complex hierarchical structures of POSIX-like file systems are often not needed [1]. Their low-level, fine-grained binary access methods provide the user with a complete control over the data layout. This enables a level of application-specific optimizations, which structured storage systems such as key-value stores or relational databases cannot provide. Such optimizations are leveraged in various contexts: Microsoft Azure [2] uses blobs for storing virtual hard disks, while in RADOS [3] they stand as a base layer for higher-level storage systems such as an object store or a distributed file system.

Yet, many of these applications also need tools to synchronize storage operations. For instance, services storing and indexing streams of events need to ensure that indexes are kept synchronized with the storage. To this end, Elastic [4] uses distributed locks to perform atomic updates to multiple indexes. Such an application-level synchronization may hinder performance while increasing development complexity.

Transactions [5] provide a simple model to cope with such data access concurrency and to coordinate writes to multiple data objects at once. We can think of three levels where to implement transactions: within applications, at middleware level or in the storage system. Application-level synchronization increases the development complexity. Also, should the application fail in the middle of a series of related updates, the storage could be left in an inconsistent state. Handling transactions at middleware level eases application development. Unfortunately, it often remains incompatible with the high performance requirements of data-intensive applications. Actually, middleware synchronization protocols come on top of those of the underlying storage, causing an increased network load and therefore a higher operation latency. In contrast, handling transactions at the storage layer enables their processing to be part of the storage operations, keeping its overhead as low as possible. This is the approach we advocate in this paper.

Although high-performance transactional distributed file systems [6] or key-value stores [7], [8] have been proposed, the underlying transaction algorithms are not easily adaptable to blobs. Indeed, existing algorithms operate on relatively small objects: records for databases and key-value stores, or on the metadata level by it from the data in distributed file systems. This separation comes at a cost: increased storage latency due to metadata lookup. Although relevant for complex hierarchical structures such as file systems, we argue that such a separation is an overkill for the flat namespace of blobs. Yet, enabling transactions on large, chunked objects is hard because of the need to ensure consistent reads across multiple chunks. In this paper we address this issue. Our contributions are:

- **We detail the challenges of integrating transactions with storage for large objects** (Section III). To that end, we consider the case of the MonALISA [9] monitoring of the ALICE [10] experiment (Section II).
- **We propose Týr, a new blob storage architecture**, leveraging the design of existing state-of-the-art storage systems (Section IV). Týr brings *lightweight multichunk and multiblob transactions under heavy access concurrency* while providing *sequential consistency guarantees*.
- **We extend this design with novel version management and read protocols** (Section V) allowing clients to read a consistent version of a blob spanning multiple chunks in presence of concurrent, conflicting writers while transparently deleting old, unused chunk versions.
- **We implement a prototype of Týr** (Section VI), that **we evaluate at large scale** on up to 256 nodes (Section VII). We show that Týr outperforms state-of-the-art solutions while providing stronger consistency guarantees.

We briefly discuss the applicability of Týr (Section VIII) and review related work (Section IX). We conclude on future work that further enhances our design (Section X).

II. A MOTIVATING USE CASE: ALICE

Týr is designed as a general-purpose storage system for a wide range of applications, such as *indexing, analytics or search services*. We illustrate its features by considering the needs of a real, production application: ALICE [10].

A. Context: big data analytics

ALICE (A Large Ion Collider Experiment) [10] is one of the four LHC (Large Hadron Collider) experiments run at CERN (European Organization for Nuclear Research) [11]. ALICE collects data at a rate of up to 16 GB/s and produces more than 10^9 data files yearly in more than 80 datacenters worldwide.

We focus on the management of the monitoring information collected in real-time about ALICE resources, which is provided by the MonALISA [9] service. More than 350 MonALISA instances are running at sites around the world, collecting information about ALICE computing facilities, network traffic, and the state and progress of the many thousands of concurrently running jobs. This yields more than 1.1 million measurements pushed to MonALISA, each with a one-second update frequency. To be presented to the end-user, the raw data is aggregated to produce about 35,000 system-overview metrics, and grouped under different time granularity levels.

B. Managing monitoring data: what could be improved?

The current implementation of MonALISA is based on a PostgreSQL database [12]. Aggregation is performed by a background worker task at regular intervals. With the constant increase in volume of the collected metrics, this storage architecture becomes inefficient. Storing each event individually, along with the related metadata, leads to a significant storage overhead. In MonALISA, the queries are well-known. This opens the way to a highly-customized data layout that would at the same time dramatically increase throughput, reduce metadata overhead, and ultimately lower both the computing and storage requirements for the cluster.

Moreover, MonALISA uses application-level locks to synchronize writing the latest measurements in presence of concurrent readers. Such pessimistic synchronization causes high lock contention, and thus reduced throughput both for clients (delayed requests) and for writing services (contradicting the real-time monitoring promise of MonALISA).

C. The need for transactions

We address these limitations by switching to a blob storage model, which better suits the needs of MonALISA. All measurements (*timestamp*, *measurement*) are appended to a *large, chunked per-generator blob*, and are averaged over a one-minute window with different granularity levels, each being stored in a different blob. This layout is explained in Figure 1.

Consequently, in addition to multichunk reads in a single blob, MonALISA requires *consistent writes across multiple blobs*. This is because adding a new measurement updates these multiple blobs simultaneously, with a byte-level granularity. To guarantee the correctness of the write and to enable hot snapshotting, such multiblob updates must be atomic.

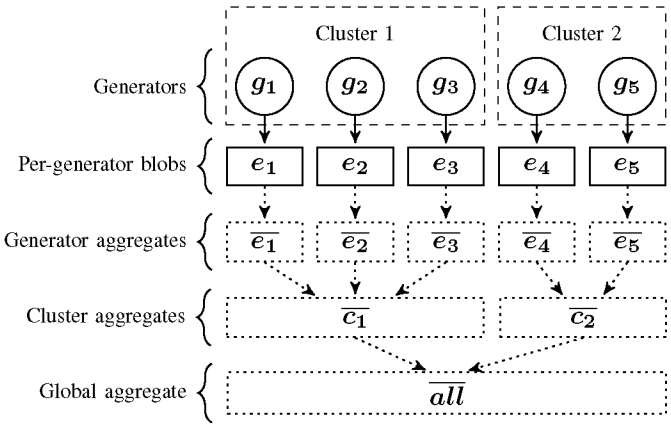


Fig. 1. Simplified MonALISA data storage layout, showing five generators on two different clusters, and only three levels of aggregation (*generator*, *cluster*, and *all*). Solid arrows indicate events written and dotted arrows represent event aggregation. Each rectangle indicates a different blob.

Updating an aggregate is a three-step operation: read old value, update it with the new data, and write the new value (*read-update-write*, or *RUW*). As an optimization, read-update-write operations should be performed in-place, *i.e.,* as a single operation involving a single client-server round-trip.

D. MonALISA: the key storage requirements

Let us summarize the key requirements of a storage system supporting high-performance data management for data-intensive large-scale applications such as MonALISA:

- **Built-in multiblob transaction support.** Applications heavily relying on data indexing as well as live computation of aggregates require a transactional storage system able to synchronize read and write operations that span multiple blobs as well as to guarantee data consistency.
- **Fine-grained random writes.** The system should support fine-grained access to blobs, and allow writes at arbitrary offsets, with a byte-level granularity.
- **In-place atomic updates.** In order to support efficient computation of aggregates and to improve the performance of read-update-write operations, the system should offer in-place atomic updates of the data, such as *add*, *subtract*, *multiply* or *divide*.
- **High-throughput under heavy concurrency.** Events in MonALISA are generated concurrently at high rate, and are accessed simultaneously by a potentially large number of clients. This calls for a storage layer able to support parallel data processing to a high degree, concurrently and on large number of nodes.

Týr addresses all these requirements, allowing it to serve efficiently the MonALISA system. However, the generic nature of such requirements do not limit Týr to this specific application, or even to data indexing and analytics. Týr enables fine-grained random writes on arbitrarily large binary objects, which makes it suitable for any application leveraging blob storage (*e.g.,* Azure Storage, RADOS).

III. The challenges we tackle

Existing distributed transaction protocols typically ensure the serializability of write operations across multiple objects whose size is small enough to fit entirely on a single server. For distributed databases or key-value stores, such algorithms operate on single records. In distributed file systems, where the size of a file can be so large that it needs to be split in multiple parts (or *chunks*), these algorithms operate on the metadata, which is orders of magnitude smaller than the data. However, this hinders performance, as the metadata servers must be included in the critical path of most read and write operations.

In contrast, the decentralized architecture of distributed key-value stores such as Dynamo [13] obviates the need for such centralized metadata management. This design results at the same time in lower throughput as any piece of data can be read directly from the server on which it is stored, and in near-linear horizontal scalability. Yet, adapting this architecture to large-scale, chunked objects is a difficult task.

A. Enabling consistent, repeatable multichunk reads

Transactions algorithms usually do not provide mechanisms to ensure the atomicity of *read* operations. This works for small objects as reading a piece of data located on a single node can be performed atomically. Yet, when dealing with large, chunked objects, we need to ensure that reads spanning multiple chunks are served with a consistent view of the object in presence of conflicting, concurrent writers.

In addition to consistent reads, transaction protocols typically provide a repeatable reads guarantee, *i.e.* the guarantee that in the context of a transaction, any object read by the client will not change if the client reads the same data again. Existing transactional systems usually provide this guarantee by caching the data on the client. Intuitively, this is not possible for larger objects, which do not fit in client memory.

An elegant solution to deal with concurrent transactions is *versioning*. As such, the read protocol must also guarantee that a blob version read in the context of any given transaction is preserved until the transaction ends. Doing so is especially hard for chunked objects stored in a decentralized system, as the chunks of a blob can be spread over multiple servers.

We address this challenge by introducing a new read protocol (Section V-C) providing consistent, repeatable reads in the context of large, chunked objects.

B. Chunk version bookkeeping

Furthermore, in highly-concurrent write-heavy scenarios, removing old, unused chunk versions as soon as possible is critical, as the number of versions to maintain for the same chunk at any given moment can be very high. Yet, such bookkeeping is very hard due to the distribution of the chunks over the cluster. Indeed, as a blob can be composed of many chunks stored on multiple servers, the information about each running transaction cannot be spread to each of these servers.

We address this issue with a novel bookkeeping method (Section V-E) that efficiently propagates information about unused chunk versions over the cluster.

IV. Background

The base architecture model of Týr is that of a replicated and decentralized key-value store similar to Dynamo [13] or Cassandra [14]. We use a lightweight chain transaction protocol to provide ACID capabilities to the system. In this section, we describe the design principles of Týr, which are largely based on state-of-the-art practices.

A. Data striping and replication

Data striping is used to balance reads and writes over a large number of nodes. Blobs are split into multiple chunks of a size defined for the whole system. With a chunk size s, the first chunk c_1 of a blob will contain the bytes in the range $[0, s)$, the second chunk c_2, possibly stored on another node, will contain the bytes in the range $[s, 2s)$, and the chunk c_n will contain the bytes in the range $[(n-1) * s, n * s)$. The chunk size is a system parameter: a typical value is 64 MB. Each chunk is replicated on multiple servers: the default replication factor is 3.

B. Distributed Hash Table based data distribution

Chunks are distributed across the cluster using consistent hashing [15], based on a *distributed hash table*, or DHT. Given a hash function $h(x)$, the output range $[h_{min}, h_{max}]$ of the function is treated as a circular space (h_{min} wrapping around to h_{max}). Every node is assigned a different random value within this range, which represents its position on the ring. For any given chunk n of a blob k, a position on the ring is calculated by hashing the concatenation of k and n using $h(k : n)$. The *primary node* holding the data for a chunk is the first one encountered while walking the ring past this position. Additional replicas are stored on servers determined by continuing walking the ring until a number of nodes equal to the replication factor are found.

C. Warp transaction protocol

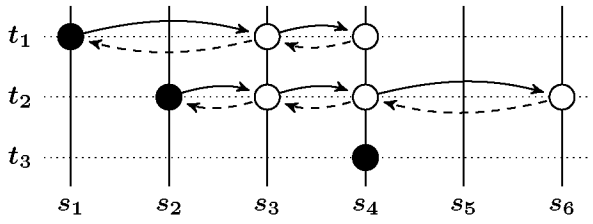
Týr uses the Warp optimistic transaction protocol, whose correctness and fault-tolerance has been proven in [7]. Warp was introduced for the HyperDex [16] key-value store, providing lightweight ACID transactions for a decentralized system. In order to commit a transaction, the client constructs a chain of servers which will be affected by it. These nodes are all the ones storing the written data chunks, and one node holding the data for each chunk read during the transaction (if any). This set of servers is sorted in a predictable order, such as a bitwise ordering on the IP/Port pair. The ordering ensures that conflicting transactions pass through their shared set of servers in the exact same order. The client addresses the request to the coordinator. This node will validate the chain and ensure that it is up-to-date according to the latest ring status. If not, that node will construct a new chain and forward the request to the coordinator of the new chain.

Warp uses a linear transactions commit protocol to guarantee that all transactions are either successful and serializable, or abort with no effect. This protocol consists of one forward pass to optimistically validate the values read by the client and ensure that they remained unchanged by concurrent transactions, followed by a backward pass to propagate the result of the transaction – either success or failure – and actually commit the changes to memory. Dependency information is embedded by the nodes in the chain during both forward and backward passes to enforce a serializable order across all transactions. A background garbage collection process limits this number of dependencies by removing those that have completed both passes.

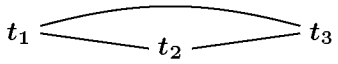
The coordinator node does not necessarily own a copy of all the chunks being read by every transaction, which are distributed across the cluster. As such, one node responsible for a chunk being read in any given transaction must validate it by ensuring that this transaction does not conflict nor invalidates previously validated transactions, for which the backward pass is not complete. Every node in the commit chain ensures that the transactions do not read values written by, or write values read by previously validated transactions. Nodes also check each value against the latest one stored in their local memory to verify that the data was not changed by a previously committed transaction. The validation step fails if transactions fail either of these tests. A transaction is aborted by sending an abort message backwards through the chain members that previously validated the transaction. These members remove the transaction from their local state, thus enabling other transactions to validate instead. Servers validate each transaction exactly once, during the forward pass through the chain. As soon as the forward pass is completed, the transaction may commit on all servers. The last server of the chain commits the transaction immediately after validating it, and sends the commit message backwards to the chain.

Enforcing a serializable order across all transactions requires that the transaction commit order does not create any dependency cycles. To this end, a local dependency graph across transactions is maintained at each node, with the vertices being transactions and each directed edge specifying a conflicting pair of transactions. A conflicting pair is a pair of transactions where one transaction writes at least one data chunk read or written by the other. Whenever a transaction validates or commits after another one at a node, this information is added to the transaction message sent through the chain: the second transaction will be recorded as a dependency of the first. This determines the directionality of the edges in the dependency graph. A transaction is only persisted in memory after all of its dependencies have committed, and is delayed at the node until this condition is met.

Figure 2 illustrates this protocol with an example set of conflicting transaction chains and the associated dependency graph. This example shows three transaction chains executing. Figure 2a shows the individual chains with the server on which they execute. The black dot represents the coordinating server for each transaction, the plain lines the forward pass, and the dashed lines the backwards pass. Because they overlap at some servers, they form conflicting pairs as shown on the dependency graph in Figure 2b. The directionality of the edges will be decided by the protocol, as the chain is executed.



(a) Propagation graph



(b) Dependency graph

Fig. 2. Warp protocol. In this example, the directionality of the edge (t_1, t_2) is decided by s_4, last common server in the transaction chains, during the backwards pass. Similarly, the directionality of (t_2, t_3) is decided by s_4.

V. TÝR: DESIGN OF A TRANSACTIONAL BLOB STORE

A. Interface of Týr

Týr provides a low-level binary API, granting users access to the data stored in blobs down to byte-level granularity. Multiblob ACID transactions enable users to commit multiple reads and writes as a single atomic operation, guaranteed to either succeed or fail as a complete unit. In-place atomic updates allow some read-update-write operations such as *add* or *subtract* to be processed directly on the server. In order to demonstrate the usage of Týr with a concrete example, we illustrate it in the context of MonALISA.

Algorithm 1 details the process of writing a new measurement to the storage system. Everything happens in the context of a single local transaction, opened by calling the BEGIN function. Inside the transaction, every write is locally recorded by the client; no information is sent to the storage system until the transaction is committed using the COMMIT function. The WRITE function (not explicitly present in this example) writes a binary value at a given offset in a blob. APPEND appends a binary value to a blob. APPLY applies an operation in-place – in our example, an arithmetic addition. Algorithm 2 reads an aggregate value. Because this operation only needs a single READ call, it does not need to be executed inside the context of a transaction. A transaction containing only read calls or only write calls is guaranteed to succeed on a healthy cluster, i.e. if the number of server failures does not exceed the system limits discussed in Section VIII.

B. Týr's high-level version management

In order to achieve high write performance under concurrent workloads, Týr uses *multiversion concurrency control* (or *MVCC*) [17]. This ensures that the current version of a blob can be read consistently while a new one is being created

Algorithm 1 Measurement update process. Functions in bold are part of the Týr API.

```
connection ← CONNECT()                          ▷ Connect to the server

▷ Save a value val at time t generated by gen in cluster
procedure SAVEMEASUREMENT(val, time, gen, cluster)
    BEGIN(connection)                           ▷ Open a transaction context
    ▷ Append new measurement to the per-generator blob
    let data be the concatenation of time and val
    APPEND(gen, data)                           ▷ Append data to gen blob
    ▷ Update aggregates
    UPDATEAGGREGATE("a_" + gen, val, time)
    UPDATEAGGREGATE("a_" + cluster, val, time)
    UPDATEAGGREGATE("a_all", val, time)
    COMMIT()                                    ▷ Atomically commit changes
end procedure

procedure UPDATEAGGREGATE(blob, val, time)
    let offset be the offset at which to write in blob
    ▷ Add 1 in place to the measurement count.
    APPLY(blob, offset, ADD, 1)
    ▷ Add the measurement to the total.
    APPLY(blob, offset + SIZEOF(int), ADD, val)
end procedure
```

Algorithm 2 Measurement read process. Functions in bold are part of the Týr API.

```
connection ← CONNECT()                          ▷ Connect to the server

procedure READAGGREGATE(blob, time)
    let offset be the offset at which to write in blob
    data ← READ(blob, offset, 2 * SIZEOF(int))
end procedure
```

without locking. Version management is done implicitly in Týr. Version identifiers are internal to Týr and are not exposed to the client.

Any given transaction is assigned a globally unique identifier at the client. During the transaction commit phase, a new version of each chunk being written to is generated on all the nodes the chunk is stored on. The new chunk version identifier is the transaction identifier. The version of all other chunks remains unchanged, as illustrated by Figure 3. In this example, the *blob version* v_6 is composed of the *chunk versions* (v_4, v_3, v_6). The blob version identifier is the same as the most recent version identifier of its chunks.

For any given write operation, only the nodes holding affected data chunks will receive information regarding this new version. Consequently, the latest version of a blob is composed of a set of chunks with possibly different version identifiers, as illustrated by Figure 4. To be able to read a consistent version of any given blob, information regarding every successive versions of the chunks composing this blob is stored on the same nodes holding replicas of the first data

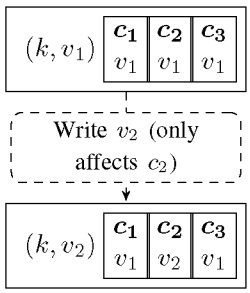


Fig. 3. Týr versioning model. When a version v_2 of the blob is written, which only affects chunk c_2, only the version of both the blob and c_2 is changed. The version id for both c_1 and c_3 remains unchanged.

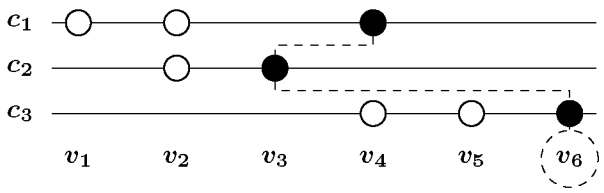


Fig. 4. Version management example. The version v_1 of this blob only affected the chunk c_1, v_2 affected both c_1 and c_2. In this example, v_6 is composed of the chunk versions (v_4, v_3, v_6). This versioning information is stored on the blob's metadata nodes.

chunk of the blob. These nodes are called *version managers* for the blob. Co-locating the first chunk of a blob and its version managers enables faster writes to the beginning of blobs. This version information placement enables the client to address requests directly to a version manager, avoids using any centralized version manager or metadata registry.

C. Týr's read protocol

1) Direct read: With chunk placement based on a distributed hash-table, clients are able to locate efficiently the nodes holding the replicas of any given data chunk. Reading the latest version of a chunk is thus straightforward: the client sends a request directly to one of these nodes. The server responds with the latest version of that chunk.

2) Consistent read: Direct reads are not applicable in a number of cases involving reading multiple chunks of a blob. When reading a portion of a blob which overlaps multiple chunks, it is necessary to obtain the version identifiers of each of those chunks forming a consistent version of the blob as explained in Section V-B. Inside a transaction, successive read operations on any given blob must be performed on the same consistent blob version, even if the portions of the blob to be read lie on different chunks.

a) In the context of a transaction: The first read operation on a blob is sent to one of its version managers, which constructs a list of the chunk version identifiers composing the latest version of the blob. For each chunk being read, the version manager randomly selects one replica and forwards the read request to the node holding it, along with the associated chunk version in the list. These nodes respond to the version manager with the requested version of the

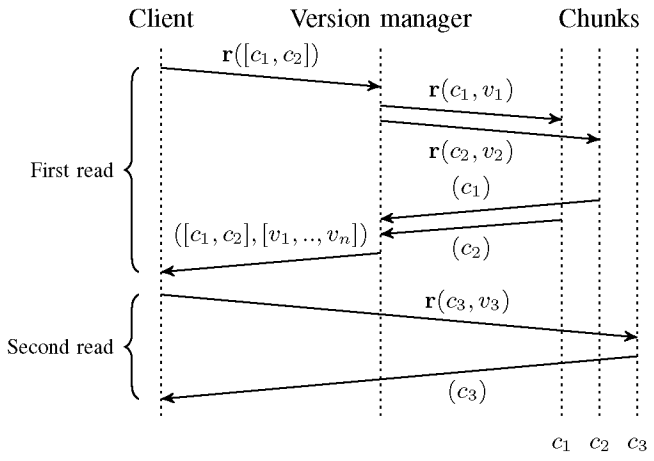


Fig. 5. Týr read protocol inside a transaction. The client sends a read query for chunks c_1 and c_2 to the version manager, which relays the query to the servers holding the chunks with the correct information, and responds to the client with the chunk data and a snapshot of the chunk versions. Subsequent read on c_3 is addressed directly to the server holding the chunk data using the received chunk version information.

data. Upon reception of the replies from each node by the version manager, it responds to the client with the received data. This message is piggybacked with the list of chunk version identifiers constructed for this request. These version identifiers are cached by the client in the transaction. Any subsequent read operation on the same blob within the same transaction can then be processed as a direct read: the client addresses the request directly to the nodes holding the chunks to be read, attaching the associated version identifier. The whole process is illustrated by Figure 5. This protocol enables the client to read a consistent version of the blob even in presence of concurrent writes to the chunks being accessed.

b) Outside of a transaction: A read operation overlapping multiple chunks is processed like the first read inside a transaction: it has to go through the version manager of the blob. Unlike a transactional read, chunk version information is not sent back to the client. Subsequent reads on the same blob may be performed on a newer version of the blob in presence of concurrent writes.

D. Handling ACID operations: Týr's write protocol

The Warp protocol briefly introduced in Section IV-C has not been designed for blob storage systems. Adapting it for Týr and coupling it with multiversion concurrency control is not a trivial task.

The version managers of a blob have complete information about the successive versions of the blob. Thus, the version managers of any given blob have to be made aware of any write to this blob. We achieve this by systematically including the version managers in the Warp commit chain, in addition to the nodes holding the chunk data. The version managers of all the blobs being written to in a given transaction are ordered using the same algorithm used for the rest of the chain, and are inserted at the beginning of the chain. This ensures that, during the backward pass of the transaction commit protocol, the transaction will have successfully committed on every chunk storage node before it is marked as committed on the version manager nodes.

Correctness: A transaction t_1 conflicts with another transaction t_2 only if it reads a chunk written to by t_2, if it writes to a chunk being read to by t_2, or if it writes to a common set of chunks. The chain commit protocol enables the first two cases to be detected during the forward pass, regardless of the chain ordering. Placing the ordered list of version managers at the beginning of the chain does not break the correctness of the chain commit algorithm: if t_1 and t_2 write to a common set of blob chunks, the version managers for these blobs will be included in both commit chains, sorted in the same order, and will be inserted at the beginning of each. Consequently, two conflicting transactions will keep the same chain relative order, as required by the commit protocol.

E. Version garbage collector

1) Garbage collector overview: Týr uses multiversion concurrency control as part of its base architecture in order to handle lock-free read / write concurrency. Týr also uses versioning to support the read protocol, specifically to achieve write isolation and ensure that a consistent version of any blob can be read even in the presence of concurrent writes. A background process called *version garbage collector* is responsible for continuously removing unused chunk versions on every node of the cluster. A chunk version is defined as unused if it is not part of the latest blob version, and if no version of the blob it belongs to is currently being read as part of a transaction.

The question now is how to determine the unused chunk versions. The transaction protocol defines a serializable order between transactions. It is then trivial for every node to know which is the last version of any given chunk it holds, by keeping ordering information between versions. Determining whether a chunk version is part of a blob version being read inside a transaction is however not trivial. Intuitively, one way to address this challenge is to make the version managers of the blob responsible for ordering chunk version deletion. This is possible because the read protocol ensures that a read operation on any given node inside a transaction will always hit a version manager of this blob. Hence, at least one version manager node is aware of any running transaction performing a read operation in the cluster. Finally, the version managers are aware of the termination of a transaction as they are part of the commit chain.

2) Detecting and deleting unused chunk versions: The key information allowing to decide which chunk versions to delete is stored by the version managers. We pose as a principle that a node will never delete any chunk version unless it has been cleared to do so by every version manager of the blob the chunk belongs to. Version managers keep a list of every running transaction for which they served as relay for the first read operation on a blob. Upon receiving a read operation from a client, a version manager increases a usage

counter on the latest version of the blob (i.e. the blob version used to construct the chunk version list as per the consistent read protocol detailed in Section V-C2). This usage counter is decremented after the transaction is committed, or after a defined read timeout for which the default is 5 minutes. Such timeout is necessary in order not to maintain blob versions indefinitely in case of a client failure. The list of currently used chunk versions of any blob is communicated to all the nodes holding the chunks for this blob by means of the transactional commit protocol: for any given transaction, for every blob being written to by the transaction, each version manager piggybacks to the forward and backward pass messages the list of chunk versions currently in use. This guarantees that every node which is part of the chain will get this chunk version usage information as part of the protocol, either during the forward or the backward pass.

For any given blob, the version garbage collector of any node is free to delete any chunk version that (1) is not the latest, (2) is older than the latest transaction for which version usage information for that blob has been received from the version managers, and (3) was not part of the chunk versions in use as part of the latest version usage information received for this blob. The version garbage collector may also safely delete any chunk version older than the read timeout which has not been superseded by a newer version.

3) Optimizing the message size: In order to limit the commit message size overhead when the number of currently used chunk versions is high, we define a threshold above which these versions are piggybacked by the version managers to the transaction messages as bloom filters [18]. A bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is member of a set. The version garbage collector can efficiently check whether a blob version is in the set of currently running transactions. Bloom filters are guaranteed never to cause false negatives, ensuring that no currently used chunk will be deleted. However, bloom filters can return false positives, which may cause a chunk version to be incorrectly considered as being used. These chunk versions will be eventually deleted during a subsequent transaction involving this blob, or once the read timeout was exceeded. The false positive probability of a bloom filter depends on its size, and is a system parameter. The default is a 0.1% error probability. With this configuration, 100 concurrent running transactions on each of the 3 version managers of a blob would cause a commit message overhead of less than 0.6 Kilobytes.

VI. Týr's prototype implementation

All the features of Týr relevant for this paper have been fully implemented. This prototype implementation includes the Týr server, an asynchronous C client library, as well as partial C++ and Java bindings. The server itself is approximately 22,000 lines of GNU C code. This section describes key aspects of the implementation.

Týr is internally structured around multiple, lightweight and specialized event-driven loops, backed by the LibUV library [19]. When a request is received, it is forwarded to one of the relevant event loops for further asynchronous processing. No request queueing is done in order to avoid communication delays, and thus reduce the overall latency of the server. On-disk data and metadata storage uses Google's LevelDB key-value store [20], a state-of-the-art log-structured merge tree [21] based library optimized for high-throughput.

The intra-cluster and client-server request/response messages are serialized with Google's FlatBuffers [22] library. It allows message serialization and deserialization without parsing, unpacking, or any memory allocation. These messages are transmitted using the UDT protocol [23]. UDT is a reliable UDP-based application level data transport protocol. UDT uses UDP to transfer bulk data with its own reliability control and congestion control mechanisms. This enables transferring data at a much higher speed than TCP. We demonstrate in Section VII-A that although UDT arguably supports the performance of Týr, it does not cause any experimental bias.

VII. Evaluation

We evaluated our design in five steps. We first studied the transactional write performance of a Týr cluster with a heavily-concurrent usage pattern. Second, we tested the raw read performance of the system. We then gauged the reader/writer isolation in Týr. Fourth, we measured the performance stability of Týr over a long period. Last, we proved the horizontal scalability of Týr.

Experimental setup. We deployed Týr on the Microsoft Azure Compute [2] platform on up to 256 nodes. For all experiments, we used `D2 v2` general-purpose instances, located in the `East US` region (Virginia). Each virtual machine has access to 2 CPU cores, 7 GB RAM and 60 GB SSD storage. The host server is based on 2.4 GHz Intel Xeon E5-2673 v3 processors [24] and is equipped with 10 Gigabit ethernet.

Evaluated systems. To the best of our knowledge, no distributed storage systems with a comparable low-level data model and built-in transactions are available today. Throughout these experiments, we compared Týr with RADOS [3], a distributed blob storage system developed as part as Ceph [25]. RADOS is based on a decentralized architecture and does not make use of Multiversion Concurrency Control. We also compared Týr with BlobSeer [26], an open-source, in-memory distributed storage system which shares the same data model and a similar API. BlobSeer has been designed to support a high-throughput for highly-concurrent accesses to shared distributed blobs. Unlike Týr, BlobSeer distributes the metadata over the cluster by means of a distributed tree. Finally, we compared Týr with Microsoft Azure Storage blobs, a fully-managed blob storage system provided as part of the Microsoft Azure cloud platform. This system comes in three flavors: *append blobs, block blobs and page blobs*. Append blobs are optimized for append operations, block blobs are optimized for large uploads, and page blobs are optimized for random reads and writes [27]. For our experiments, we used RADOS 0.94.4 (Hammer), BlobSeer 1.2.1, and the version of Azure Storage Blobs available at the time these experiments were run.

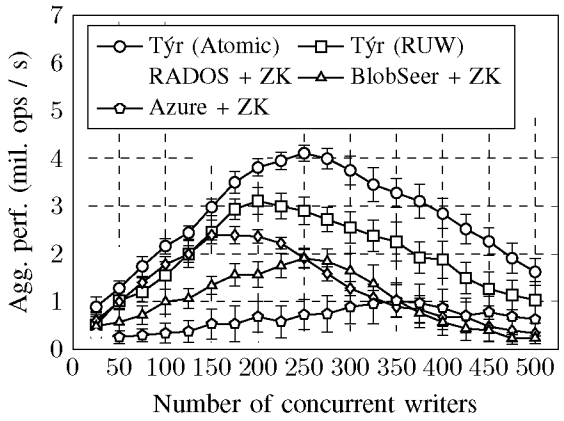


Fig. 6. Transactional write performance of Týr, RADOS, BlobSeer and Azure Storage, varying the number of clients, with 95% confidence interval.

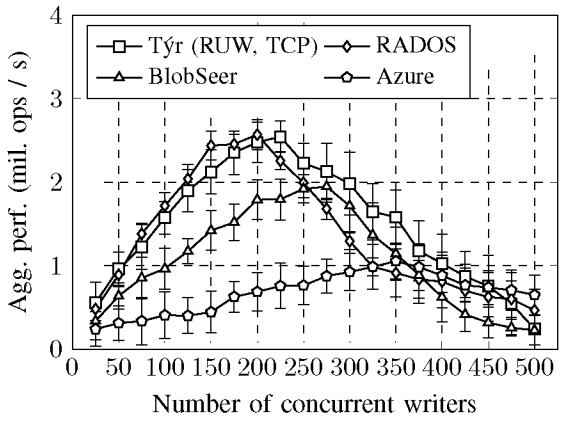


Fig. 7. Write performance of Týr (without UDT), RADOS, BlobSeer and Azure Storage, varying the number of clients, with 95% confidence interval.

Dataset and workload. In order to run these experiments, we used a dump of real data obtained from the MonALISA system [9]. This data set is composed of $\sim$ 4.5 million individual measurement events, each one being associated to a specific monitored site. We used multiple clients to replay these events, each holding a different portion of the data. Clients are configured to loop over the data set to generate more events when the size of the data is not sufficient. The data was stored in each system following the layout described in Section II. The read operations were performed by querying ranges of data, simulating a realistic usage of the MonALISA interface. In order to further increase read concurrency, the data was queried following a power-law distribution.

Experimental configuration. Because of the lack of native transaction support in Týr competitors, we used ZooKeeper 3.4.8 [28], an industry-standard, high-performance distributed synchronization service, which is part of the Hadoop [29] stack. We only use ZooKeeper locks, in order to synchronize writes to the data stores. ZooKeeper locks are handled at the lowest-possible granularity: one lock is used for each aggregate offset (8-byte granularity), except for Azure in which we had to use coarse-grained locks (512-byte granularity). This is because Azure page blobs, which we used for storing the aggregates, requires writes to be aligned on a non-configurable 512-byte page size [27]. We have used page blobs because of their random write capabilities. Furthermore, append blobs are not suited for operating on small data objects such as MonALISA events: they are limited to 50.000 appends. We discuss the choice of ZooKeeper in Section VIII-A, and demonstrate in Section VII-A that this choice does not cause any bias in our experiments.

A. Write performance

High transactional write performance is the key requirement that guided the design of Týr. To benchmark the different systems in this context, we measured the transactional write performance of Týr, RADOS, BlobSeer and Azure Storage with the MonALISA workload. We also measured the performance of the same workload on the Azure Storage platform. Týr uses atomic operations. However, being the only system to support such semantics, we also tested the Týr behavior with regular read-update-write (RUW) operations as a baseline. Týr transactions are required to synchronize the storage of the events and their indexing in the context of a concurrent setup. We used ZooKeeper to synchronize writes on the other systems. All systems were deployed on a 32-node cluster, except for Azure Storage which does not offer the possibility to tune the number of machines in the cluster.

The results, depicted in Figure 6, show that the Týr peak throughput outperforms its competitors by 78% while supporting higher concurrency levels. Atomic updates allowed Týr to further increase performance by saving the cost of read operations for simple updates. The significant drop of performance in the case of RADOS, Azure Storage and BlobSeer at higher concurrency levels is due to the increasing lock contention. This issue appears most frequently on the global aggregate blob, which is written to for each event indexed. In contrast, our measurements show that Týr's performance drop is due to CPU resource exhaustion. Under lower concurrency, however, we can see that the transaction protocol incurs a slight processing overhead, resulting in a comparable performance for Týr and RADOS when the update concurrency is low. BlobSeer is penalized by its tree-based metadata management which incurs a non-negligible overhead compared to Týr and RADOS. Overall, Azure shows a lower performance and higher variability than all systems. At higher concurrency levels however, Azure performs better than both RADOS and BlobSeer. This could be explained by a higher number of nodes in the Azure Storage cluster, although the lack of visibility into its internals doesn't allow us to draw any conclusive explanation. We can observe the added value of in-place atomic operations in the context of this experiment, which enables Týr to increase its performance by 33% by avoiding the cost of read operations for simple updates.

Fairness: We argue that neither the choice of ZooKeeper nor the use of UDT do bias our experiments. We prove this

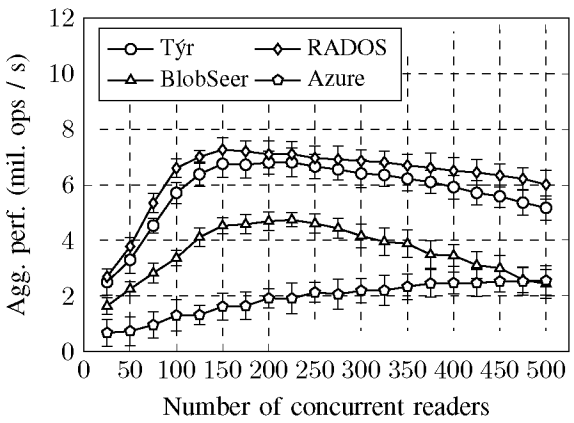


Fig. 8. Read performance of Týr, RADOS, BlobSeer and Azure Storage, varying the number of concurrent clients, with 95% confidence interval.

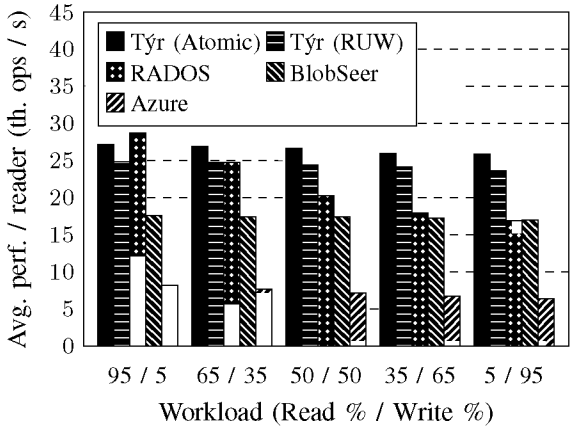


Fig. 9. Read throughput of Týr, RADOS, BlobSeer and Azure Storage for workloads with varying read to write ratio. Each bar represents the average read throughput of 200 concurrent clients averaged over one minute.

formally by running another experiment with the same parameters as above, disabling ZooKeeper for RADOS, BlobSeer and Azure Storage at the expense of write consistency, and replacing UDT by TCP in Týr. We plot the results in Figure 7, which show that Týr performance is similar to the one RADOS, while still providing significantly stronger consistency guarantees. These results are explained by the lock-free design of Týr which supports its performance in highly-concurrent workloads, and by its decentralized architecture which ensures adequate load distribution across the cluster.

B. Read performance

We evaluated the read performance of a 32-node Týr cluster and compared it with the results obtained with RADOS and BlobSeer on a similar setup. As a baseline, we measured the same workload on the Azure Storage platform. We preloaded in each of these systems the whole MonALISA dataset, for a total of around 100 Gigabytes of uncompressed data. We then performed random reads of 800 Byte size each from both the raw data and the aggregates, following a power-law distribution to increase read concurrency. This read size corresponds to a 100-minute average of MonALISA aggregated data. To prevent memory overflows, we throttle the number of concurrent requests in the system to a maximum of 1,000.

We plotted the results in Figure 8. The lightweight read protocol of both Týr and RADOS allows them to exhaust the CPU resources quickly and to outperform both BlobSeer and Azure Storage peak throughput by 44%. On the other hand, BlobSeer requires multiple hops to fetch the data in the distributed metadata tree. This incurs an additional networking cost that limits the total performance of the cluster. Under higher concurrency, we observe a slow drop in throughput for all the compared systems except for Azure Storage due to the involved CPU in the cluster getting overloaded. Once again, linear scalability properties of Azure could be explained by the higher number of nodes in the cluster, although this can't be verified because of the lack of visibility into Azure internals.

Týr and RADOS show a similar performance pattern. Measurements show RADOS outperforming Týr by a margin of approximately 7%. This performance penalty can partly be explained by the overhead of the multiversion concurrency control in Týr, enabling it to support transactional operations.

C. Reader/writer isolation

We performed simultaneously reads and writes in a 32-node cluster, using the same setup and methodology as with the two previous experiments. To that end, we preloaded half of the MonALISA dataset in the cluster and measured read performance while concurrently writing the remaining half of the data. We ran the experiments using 200 concurrent clients. With this configuration, all three systems proved to perform above 85% of their peak performance for both reads and writes, thus giving comparable results and a fair comparison between the systems. Among these clients, we varied the ratio of readers to writers in order to measure the performance impact of different usage scenarios. For each of these experiments, we were interested in the average throughput per reader.

The results, depicted in Figure 9, illustrate the added value of *multiversion concurrency control*, on which both Týr and BlobSeer are based. For these two systems, we observe a near-stable average read performance per client despite the varying number of concurrent writers. In contrast, RADOS, which outperforms Týr for a 95/5 read-to-write ratio, shows a clear performance drop as this ratio decreases. Similarly, Azure performance decreases as the number of writers increases.

D. Performance stability

Týr data structures have been carefully designed so that successive writes to the cluster impact access performance as little as possible. We validated this behavior over a long period of time using a 32-node cluster, and 200 concurrent clients. We used a 65% read / 35% write workload, the most common workload encountered in MonALISA.

We depict in Figure 10 the aggregated read / write throughput over an extended period of time. The results confirm

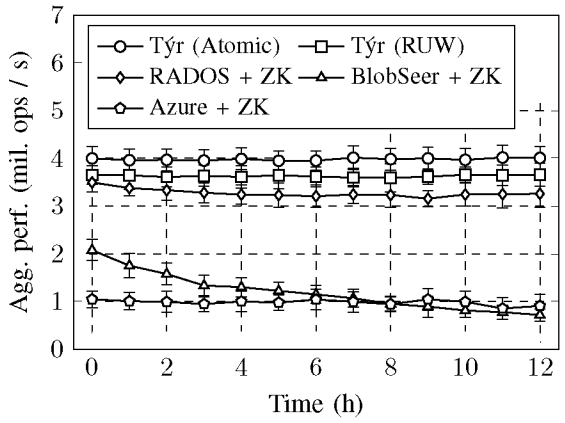


Fig. 10. Comparison of aggregated read and write performance stability over a 12-hour period with a sustained 65% read / 35% write workload, with 95% confidence intervals.

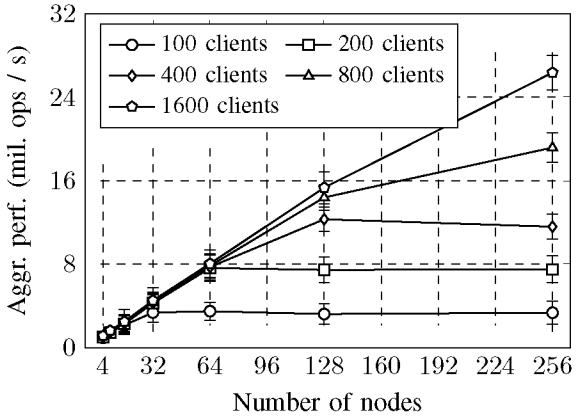


Fig. 11. Týr *horizontal scalability*. Each point shows the average throughput of the cluster over a one-minute window with a 65% read / 35% write workload, and 95% confidence intervals.

that Týr performance is stable over time. On the other hand, BlobSeer shows a clear performance degradation over time, which we attribute to its less efficient metadata management scheme: for each blob, metadata is organized as a tree that is mapped to a distributed hash table hosted by a set of metadata nodes. Accessing the metadata associated with a given blob chunk requires the traversal of this tree; as the height of this tree increases, the number of requests necessary to locate the relevant chunk metadata also increases. This results in a higher number of client-server round-trips, and consequently in a degraded performance over time. Under the same conditions, RADOS and Azure Storage showed a near-stable performance, which allows us to dismiss ZooKeeper influence in the performance decrease observed with BlobSeer.

E. Horizontal scalability

Finally, we tested the performance of Týr when increasing the cluster size up to 256 nodes. This results in an increased throughput as the load is distributed over a larger number of nodes. We used the same setup as for the previous experiment, varying the number of nodes and the number of clients, and plotting the achieved aggregated throughput among all clients over a one-minute time window. We have used the same 35% write / 65% read workload (with atomic updates) as in the previous experiment. Figure 11 shows the impact of the number of nodes in the cluster on system performance. We see that the maximum average throughput of the system scales near-linearly as new servers are added to the cluster.

VIII. Discussion

A. Experimental methodology

We believe the systems we have chosen are representative of the state-of-the-art unstructured storage systems for clouds. Unfortunately, we were not able to compare our approach to recently-introduced transactional file systems: to the best of our knowledge, none of them are released as open-source today. Such systems are presented in Section IX. Unlike other systems, Azure Storage does not provide fine-grained write access to blobs: writes need to be aligned on a 512-byte page size. Although the added overhead arguably handicaps this system in our tests, Azure Storage is the only available cloud-based unstructured storage system to offer random writes.

This lack of transactional support on the systems we compared Týr to forced us to use an external service for synchronizing write operations. Throughout our experiments, we measured the relative impact of this choice. The results show that ZooKeeper accounts for less than 5% of the request latency for write operations. Although faster, more optimized distributed locks such as Redis [30] may be available, this is unlikely to have had a significant impact on the results. We considered a transactional middleware to replace locks with a faster alternative. Although general-purpose transactional middleware such as [31] exist in the literature, the few available systems we could find are targeted at SQL databases.

B. Transaction concurrency control

Týr transaction support is based on an optimistic protocol, enabling high performance under workloads with low data contention. On workloads with higher data contention however, this design could lead to higher aborted transactions due to conflicting writes, ultimately decreasing the throughput of the cluster. Future work will focus on analyzing the performance of a Týr cluster under such workloads.

C. Týr for small and large blobs

We have designed Týr so it could cope with arbitrarily large objects. Týr is well-suited for large blobs: chunking allows to efficiently distribute writes over multiple nodes in the cluster. However, since the version management servers are unique in a blob, they could become a bottleneck if a large number of clients concurrently access a relatively small number of blobs. Applications should be designed accordingly. Týr can also cope very efficiently with small objects: its versioning scheme has been specifically designed to keep the storage overhead as low as possible, while co-locating the first data chunk and the

version management helps reducing this overhead even further. Not required by the MonALISA use case, the evaluation of this aspect has been left for future work.

IX. Related work

Extensive research efforts have been dedicated to optimize the storage of Big Data (with sizes in the order of Giga- and Terabytes), there has been relatively less progress on identifying, analyzing and understanding the challenges of ensuring increased flexibility and expressivity at the storage level (e.g. enabling transactions, general data models, support for Small Data etc.). A new generation of storage and file systems as well as optimizations brought to existing ones, try to address these new challenges. However, as described in this section, they focus on very specific issues, which they alleviate in isolation, in most cases trading performance for expressivity.

A. Blob storage systems

As the data volumes handled by modern large-scale applications is growing, it becomes increasingly important to minimize as much as possible the metadata overhead incurred by traditional storage systems such as relational databases or POSIX file systems. Twitter's Blobstore [32] and Facebook's f4 [33] storage systems are designed for immutable data, neither of those enable modifying the stored data. RADOS [3], upon which Ceph [25] is based, is a highly scalable distributed storage system from which Týr took great inspiration and with which it shares a similar data model. In contrast with Blobstore and f4, RADOS blobs are mutable, with fine-grained data access. However, unlike Týr, RADOS does not focus on providing transactional semantics, or in-place atomic updates. BlobSeer [26] introduces several optimizations. A versioning-based concurrency control enables writes in blobs at arbitrary offsets under high concurrency, effectively pushing the limits of exploiting parallelism at data-level even further. A centralized version manager is used to coordinate the writes to the cluster. The metadata is decentralized and disseminated in the cluster using a distributed tree. While this effectively helps increasing the scalability of the cluster, the throughput of the system may be decreased by the metadata tree traversal in order to locate the data in the cluster. Týr further increases both the read and write performance by eliminating at the same time the centralized version manager and the distributed metadata tree. The Warp protocol effectively permits write coordination without the need for a centralized server. Both data and metadata can be localized by clients without any network communication in most cases, additionally enabling single-hop reads in some cases.

B. Key-Value Stores

Although they expose a less flexible data model than Týr, key-value stores have greatly inspired the design of its internals. Since disk-oriented approaches to online storage are unable to scale gracefully to meet the needs of data-intensive applications, and improvements in their capacity have far outstripped improvements in access latency and bandwidth, recent solutions are shifting the focus to random access memory, with disk relegated to a backup/archival role. Most open source key-value stores have roots in work on distributed data structures [34] and distributed hash tables [15]. Similarly to Týr, they are inspired by the ring-based architecture of Dynamo [13], a structured overlay with at most one-hop request routing, in order to offer scalability and availability properties that traditional database systems simply cannot provide. Yet these properties come at a substantial cost: the consistency guarantees are often quite weak while random writes or even appends are not supported. Write operations in Dynamo require a read to be performed for managing the vector clock scheme used for conflict resolution, which proves to be a very limiting factor for high write throughput. BigTable [35] provides both structure and data distribution but relies on a distributed file system for its durability. Cassandra [14] takes inspiration from both BigTable (for the API) and Dynamo (for data distribution), and adds limited support for transactions, trading isolation and atomicity for performance. Hyperdex [16] is a key-value store in which objects with multiple attributes are mapped into a multidimensional hyperspace, instead of a ring. Hyperdex uses Warp [7] as an additional layer for providing ACID-like transactions on top of the store with minimal performance degradation. However, HyperDex is a higher-level system that offers a lower control over the data layout without support for mutable objects.

C. Distributed sile systems

Specialized file systems specifically target the needs of data-intensive applications. Ceph [25] builds upon RADOS in order to provide a distributed file system interface. Similarly to Týr, its design allows for high-performance single-hop reads. However, Ceph does not support built-in support for transactions. CalvinFS [36] uses hash-partitioned key-value metadata across geo-distributed datacenters to handle small files, with operations on file metadata transformed into distributed transactions. However, in contrast to Týr, this system only allows operations on single files. Multifile operations require transactions. The underlying metadata database can handle such operations at high throughput, but the latency of such operations tends to be higher than in traditional distributed file systems. The Warp Transactional Filesystem (WTF) [6] is a transactional file system based on the HyperDex key-value store for metadata storage, and uses Warp as its transactional protocol. WTF handles transaction processing at the metadata level. Its design focused on providing an API allowing users to construct files from the contents of other files without data copy or relocation. WTF is based on metadata servers to locate data in the cluster, consequently not providing single-hop reads. Finally, WTF does not provide the atomic operations supported by Týr's design. Such atomic operations have previously been proposed in the literature for distributed file systems by [37], but this work unfortunately does not focus on integrating them with a transactional storage system, and did not base its evaluation on a real file system or use case.

X. Conclusion and Future Work

The lack of support for transaction semantics in current distributed blob stores raises a challenge. Complex, large-scale distributed applications have no easy tool to manage related streams and sets of data, and keep them synchronized with their corresponding indexes. The goal of this paper is to fill this gap by introducing Týr, a high-performance blob storage system providing built-in multiblob transactions. It enables applications to operate on multiple blobs atomically without complex application-level coordination, while providing sequential consistency under heavy access concurrency. We evaluate Týr at large scale on up to 256 machines using a real-world use case from the CERN LHC on the Microsoft Azure cloud: its throughput outperforms that of state-of-the-art systems by more than 75%. We show that Týr scales on large clusters of commodity machines to support millions of concurrent read and write operations per second.

In order to achieve such performance, Týr leverages the benefits of a flat namespace. As such storage model also seems to fit the needs of HPC applications, we plan to investigate the applicability of Týr for campaign storage on HPC platforms.

Finally, we plan to evaluate the behavior, performance and applicability of Týr when facing different applications and workloads spanning a large spectrum of data access patterns.

XI. Acknowledgements

This work is part of the BigStorage project (H2020-MSCA-ITN-2014-642963), funded by the European Union under the Marie Skłodowska-Curie Actions. It is also supported by the ANR OverFlow project (ANR-15-CE25-0003) and the Microsoft Research - Inria Joint Centre. The experiments presented in this paper were carried out using the Azure platform provided by Microsoft in the framework of the Z-CloudFlow project and of the Azure for Research program. The authors would like to thank Costin Grigoras (CERN) for his valuable support and insights on the MonALISA monitoring traces of the ALICE experiment, as well as the anonymous reviewers who provided incredibly valuable counsel and expertise.

References

[1] M. Seltzer *et al.*, "Hierarchical file systems are dead," in *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, ser. HotOS'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–1.

[2] "Microsoft Azure," https://azure.microsoft.com/en-us/, 2016.

[3] S. A. Weil *et al.*, "RADOS: A scalable, reliable storage service for petabyte-scale storage clusters," in *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, ser. PDSW '07. New York, NY, USA: ACM, 2007.

[4] "Elastic (formerly ElasticSearch)," 2016, https://www.elastic.co/.

[5] J. Gray, "The transaction concept: Virtues and limitations (invited paper)," in *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, ser. VLDB '81. VLDB Endowment, 1981, pp. 144–154.

[6] R. Escriva and E. G. Sirer, "The design and implementation of the warp transactional filesystem," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 469–483.

[7] R. Escriva, B. Wong, and E. G. Sirer, "Warp: Lightweight multi-key transactions for key-value stores," Cornell University, Tech. Rep., 2013.

[8] S. Das *et al.*, "G-store: A scalable data store for transactional multi key access in the cloud," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 163–174.

[9] I. Legrand *et al.*, "MonALISA: An agent based, dynamic service system to monitor, control and optimize distributed systems," *Computer Physics Communications*, vol. 180, no. 12, pp. 2472 – 2498, 2009.

[10] T. A. Collaboration, K. Aamodt *et al.*, "The alice experiment at the cern lhc," *Journal of Instrumentation*, vol. 3, no. 08, p. S08002, 2008. [Online]. Available: http://stacks.iop.org/1748-0221/3/i=08/a=S08002

[11] "CERN," 2015, http://home.cern/.

[12] K. Douglas and S. Douglas, *PostgreSQL*. Thousand Oaks, CA, USA: New Riders Publishing, 2003.

[13] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

[14] A. Lakshman *et al.*, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[15] D. Karger *et al.*, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97. New York, NY, USA: ACM, 1997.

[16] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: A distributed, searchable key-value store," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 25–36.

[17] P. A. Bernstein *et al.*, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, Jun. 1981.

[18] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[19] "LibUV," 2015, https://github.com/libuv/libuv.

[20] "LevelDB," 2015, https://github.com/google/leveldb.

[21] P. O'Neil, E. Cheng *et al.*, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[22] "FlatBuffers," 2015, https://google.github.io/flatbuffers/index.html.

[23] "PanFS," 2015, http://www.panasas.com/products/panfs.

[24] "Intel Xeon E5-2673v2 processor," 2015, http://ark.intel.com/products/79930/Intel-Xeon-Processor-E5-2673-v2-25M-Cache-3_30-GHz.

[25] S. A. Weil *et al.*, "Ceph: A scalable, high-performance distributed file system," in *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 307–320.

[26] B. Nicolae, G. Antoniu *et al.*, "BlobSeer: Next-generation data management for large scale infrastructures," *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, pp. 169 – 184, 2011.

[27] "Understanding Block Blobs, Append Blobs, and Page Blobs," 2015, https://msdn.microsoft.com/en-us/library/azure/ee691964.aspx.

[28] P. Hunt, M. Konar *et al.*, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.

[29] "Apache Hadoop," 2015, https://hadoop.apache.org.

[30] "Distributed locks with Redis," 2016, http://redis.io/topics/distlock.

[31] R. Jiménez-Peris, M. Patiño-Martínez *et al.*, "Cumulonimbo: A highly-scalable transaction processing platform as a service." *ERCIM News*, vol. 89, no. null, pp. 34–35, April 2012.

[32] "Blobstore: Twitter's in-house photo storage system," 2012, https://blog.twitter.com/2012/blobstore-twitter-s-in-house-photo-storage-system.

[33] S. Muralidhar, W. Lloyd *et al.*, "f4: Facebook's warm blob storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 383–398.

[34] C. Ellis, "Distributed data structures: A case study," *Computers, IEEE Transactions on*, vol. C-34, no. 12, pp. 1178–1185, Dec 1985.

[35] F. Chang, J. Dean *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[36] A. Thomson and D. J. Abadi, "CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 1–14.

[37] P. Carns *et al.*, "A case for optimistic coordination in hpc storage systems," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, ser. SCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 48–53.